CSCE 735: Parallel Computing

Major Project

Name: Shweta Sharma
UIN: 433003780

1. (75 points) In this project, you have to develop a parallel implementation of Strassen's recursive algorithm for computing matrix-multiplications. You should choose one of the following strategies.
a. Develop a shared-memory code using OpenMP, or
b. Develop a CUDA-based code for the GPU.
Your code should compute the product of two matrices of size n x n, where $n=2^k$, and k is an input to your program. Your code should also accept k' as an input to allow the user to vary the terminal matrix size.

Ans: Our project involved developing a parallel implementation of Strassen's recursive algorithm using OpenMP, which utilizes a shared memory approach. This approach offers the advantage of using explicit directives for synchronization and parallel regions, with an implied barrier at the end of each parallel region. The user is required to input three parameters: k, k', and t. The parameter k represents the size of the matrix in powers of 2 (i.e., for a matrix of size n, n = 2^k). The parameter k' represents the maximum power of 2 at which the recursive approach is used, up to matrix size s, where s = 2^k'. Beyond s = 2^k'(s=2^k to s=1), standard multiplication is used. The third parameter, t, is the number of threads specified by the user, which allows us to control the degree of parallelization of the algorithm.

2. (25 points) Develop a report that describes the parallel performance of your code. You will have to conduct experiments to determine the execution time for different values of k, and use that data to plot speedup and efficiency graphs. You should also experiment with different values of k' to explore its impact on the execution time.
Discuss any design choices you made to improve the parallel performance of the code and how it relates to actual observations. Include any insights you obtained while working on this project.
Lastly, include a brief description of how to compile and execute the code on platform you have chosen.

Ans: To analyze the parallel performance of plots of execution time vs no of threads, speed up vs no of threads and efficiency vs no of threads is used.

Speed Up = Execution time with single thread/ Execution time with t threads.

Efficiency = Speed up/ Number of threads used.

Below are the 2 major approaches used to analyze the parallel performance using k, k‘ and number of threads:

- Fixed k' and vary the matrix size by changing k from small values like k -4 to k – 8, 9, 10 and 11.

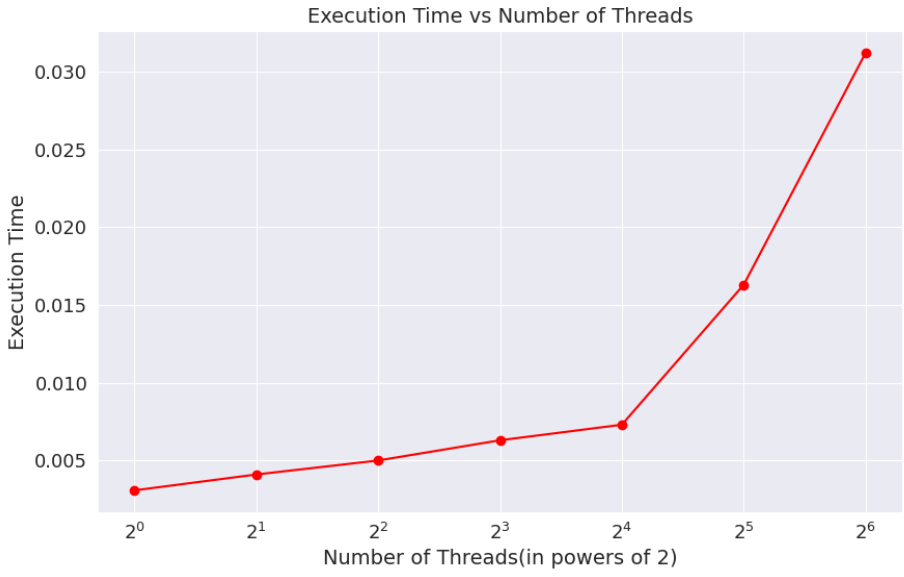
- Fixed k and changing the value of terminal matrix k’ from k’-2 ,4,5 to k’-6

Due to the recursive nature of Strassen's Algorithm and its reliance on function calls to compute M1, M2, and so on, we noticed that increasing the number of threads did not result in a significant improvement in parallelism or speedup for smaller matrix sizes (i.e., k-4). This may be due to the fact that memory usage becomes too intensive for smaller matrices, making it less practical to use the algorithm for such sizes. In fact, we observed a decrease in speedup as the parallel performance was not efficient for these smaller sizes. This is illustrated in the results below:

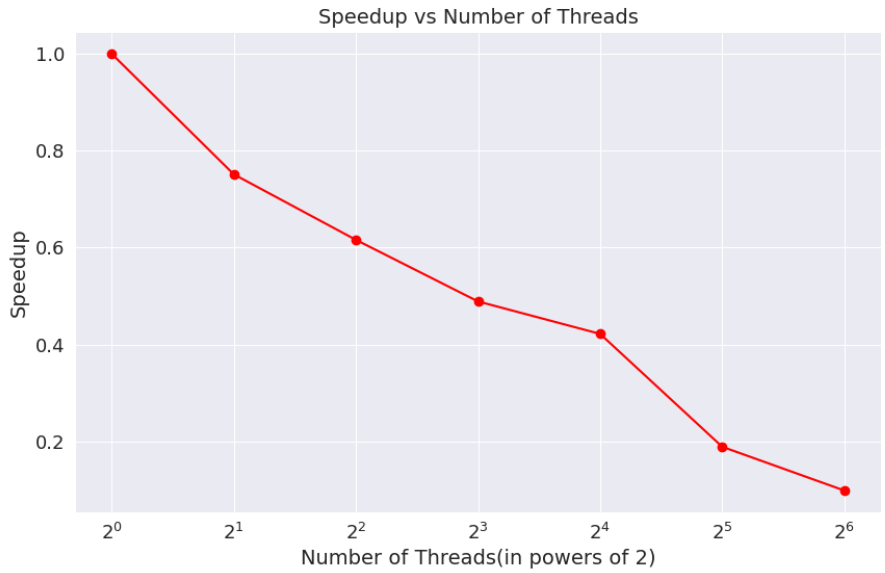
k	k'	thread	matrix size	execution time
4	2	1	16	0.00308
4	2	2	16	0.0041
4	2	4	16	0.005
4	2	8	16	0.0063
4	2	16	16	0.0073
4	2	32	16	0.0163
4	2	64	16	0.0312

thread	speed up	efficiency
1	1	1
2	0.751219512	0.375609756
4	0.616	0.154
8	0.488888889	0.061111111
16	0.421917808	0.026369863
32	0.188957055	0.005904908
64	0.098717949	0.001542468

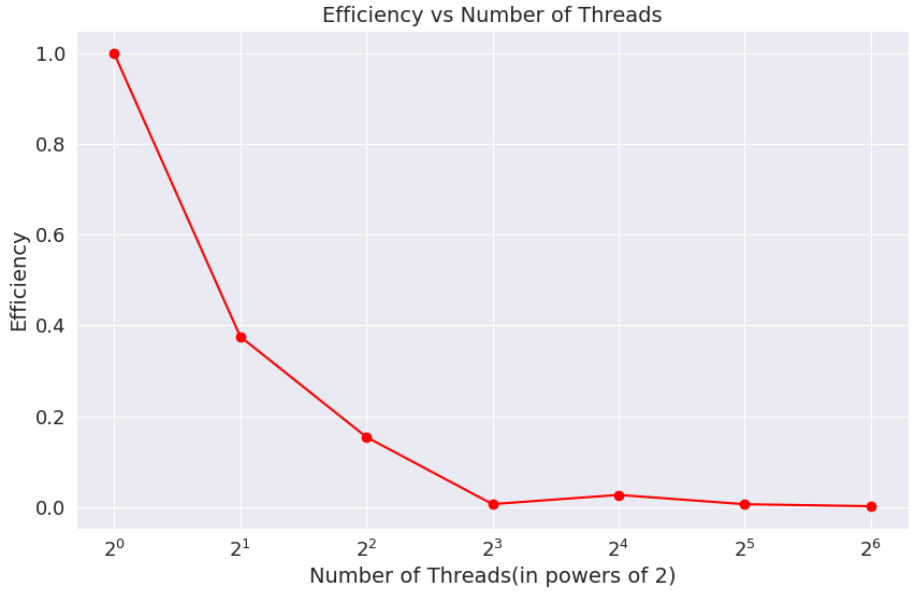
The plot for Execution time vs Number of threads look as follows:



The plot for Speedup vs Number of threads look as follows:



The plot for Efficiency vs Number of Threads look as follows:

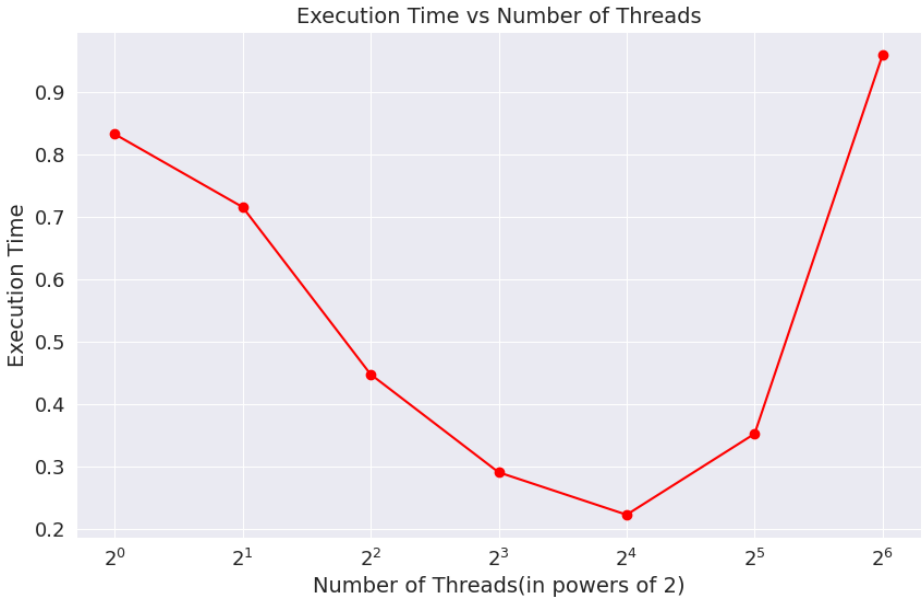


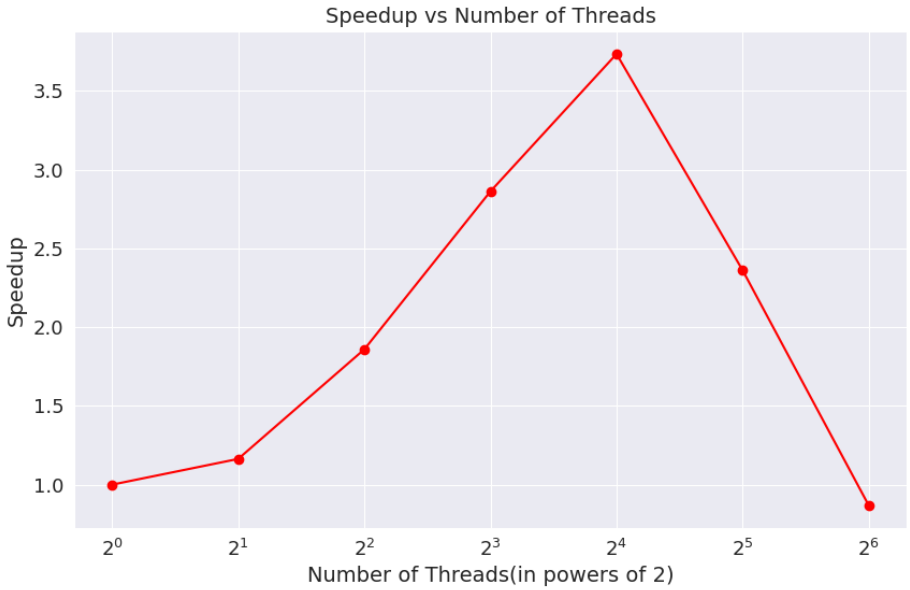
As seen, there is a gradual decrease in speed up but the efficiency also decreases as it is calculated based on the number of threads at each stage and is very less and not desirable.

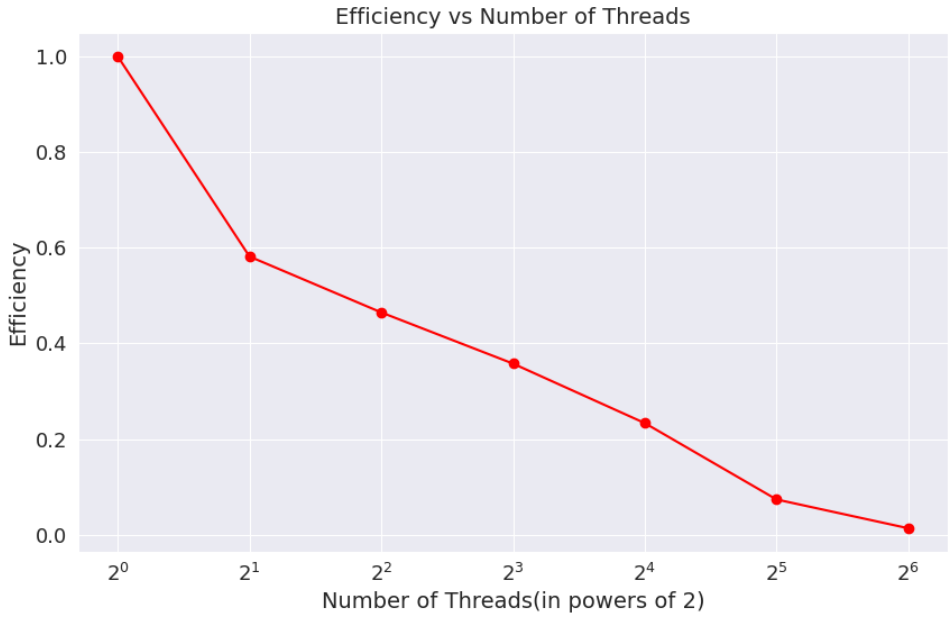
In order to achieve better parallel performance, we proceeded to implement the parallel code on larger matrix sizes, specifically for values of k equal to 8, 9, 10, and 11. To keep the range of k' consistent, we

used a smaller value of k' equal to 2, and then increased it to 6. By increasing the number of threads by a factor of 2 at every iteration, from 2^0 to 2^6, we were able to achieve the desired level of parallelism

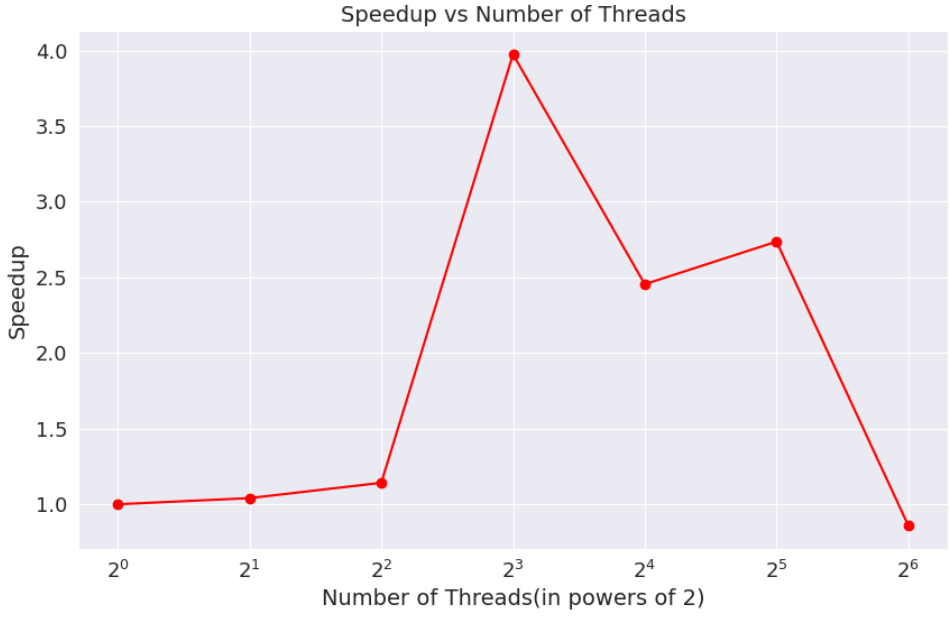
Following are the Execution Time, Speed up and efficiency plots for k = 8 and k' = 2

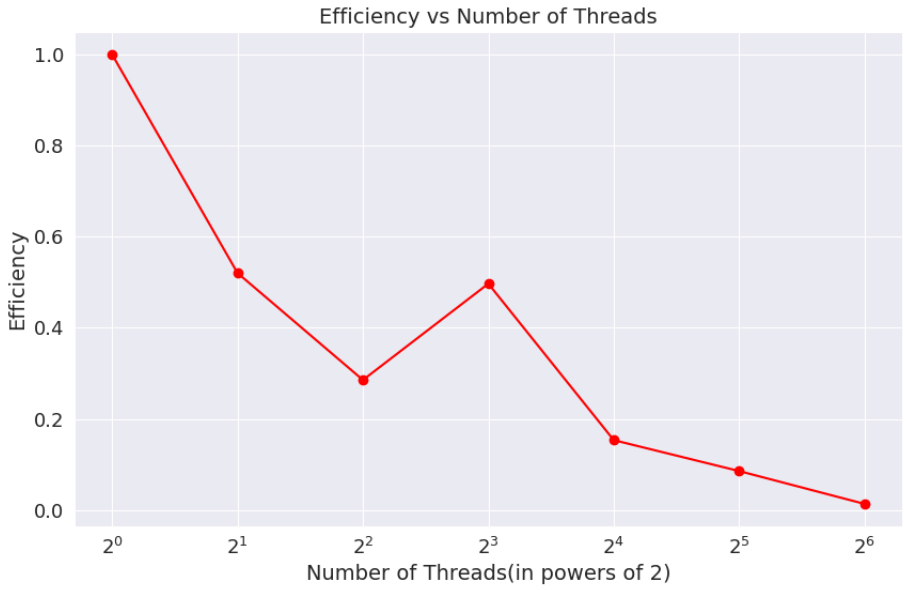




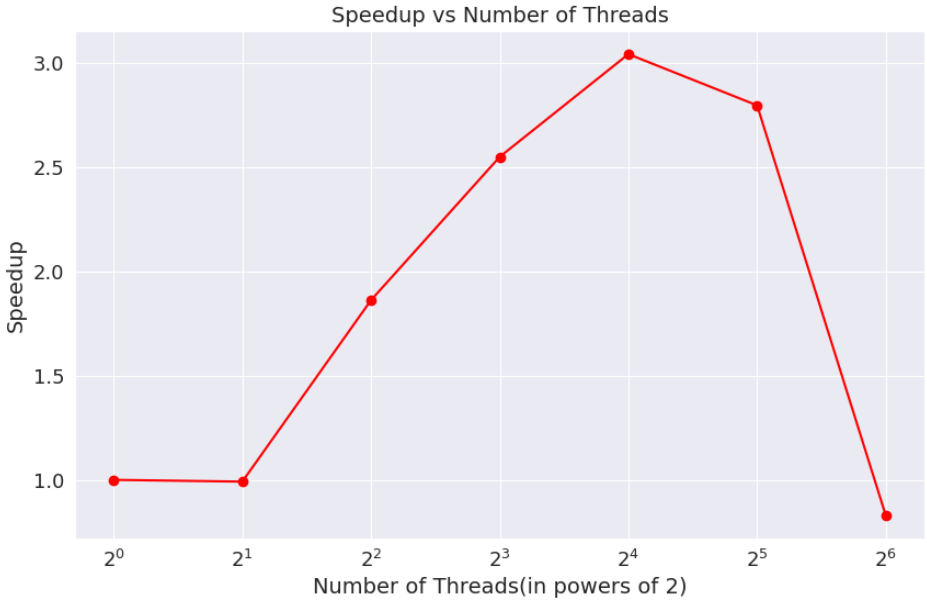


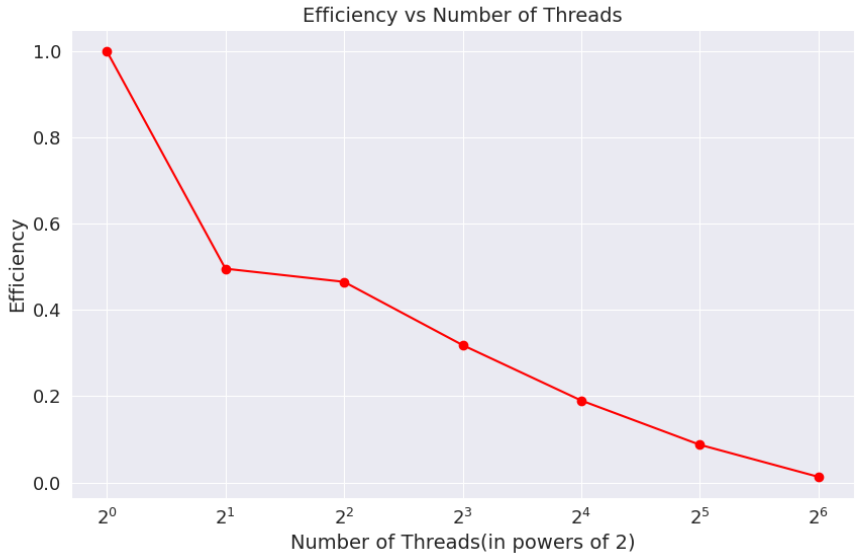
- **Following are the Speed up and efficiency plots for k = 9 and k' = 2**





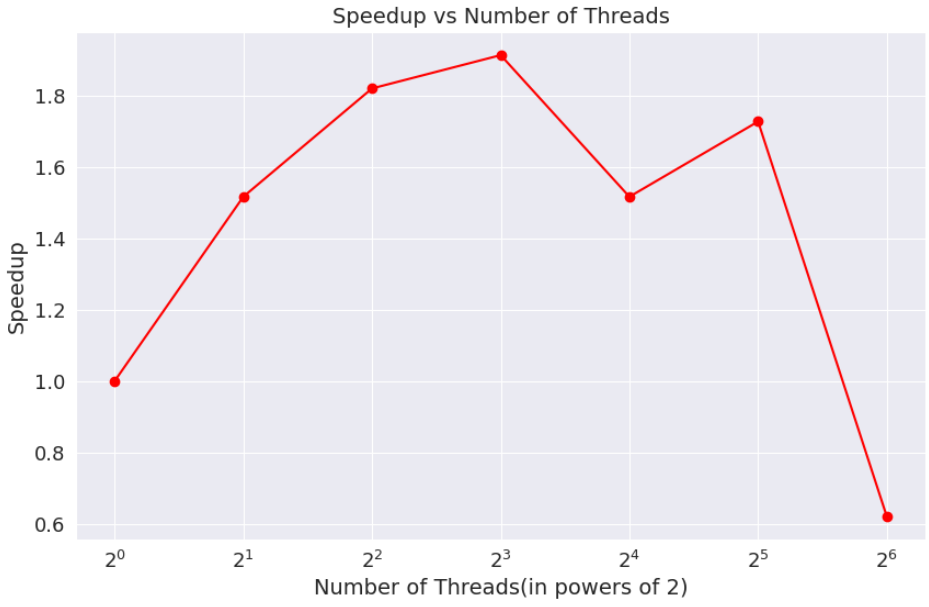
- **Following are the Speed up and efficiency plots for k = 10 and k' = 2**

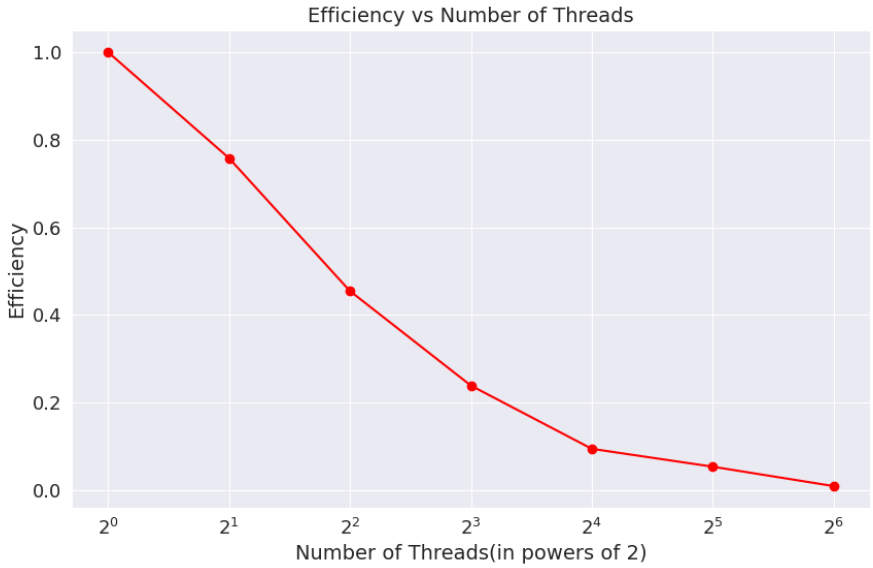




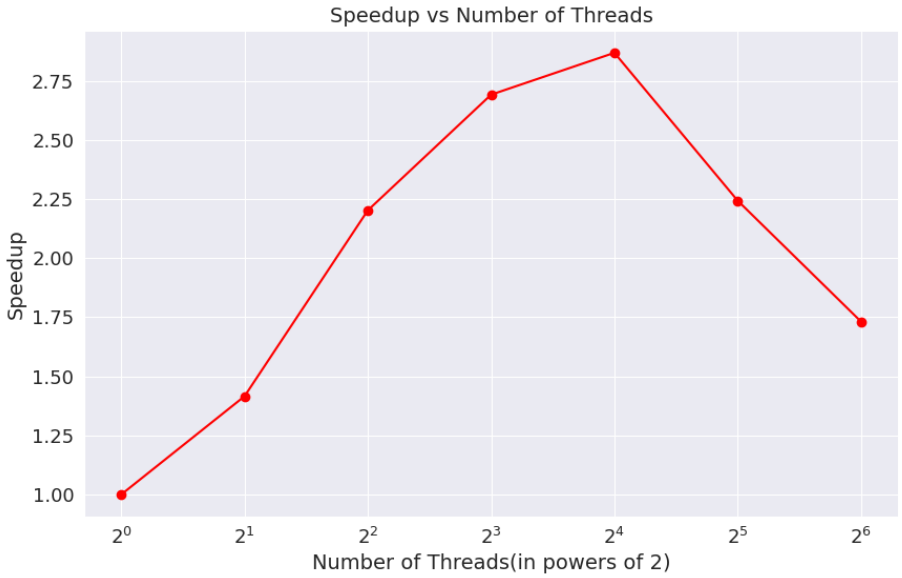
In this section again experiment was conducted by varying the k from k = 8, 9, 10, 11 with a different value for k' = 6 for better validation of speed up improvement with k size.

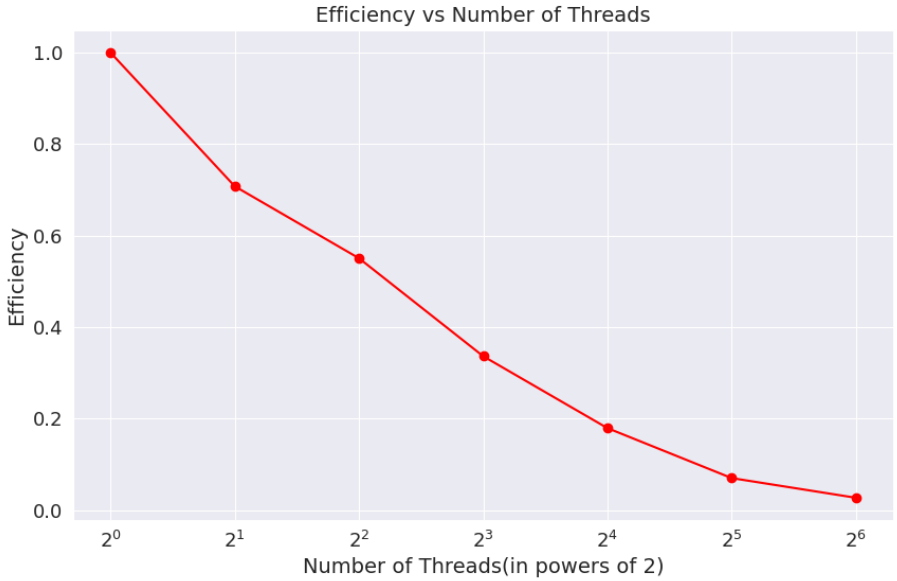
- **Following are the Speed up and efficiency plots for k = 8 and k' = 6**



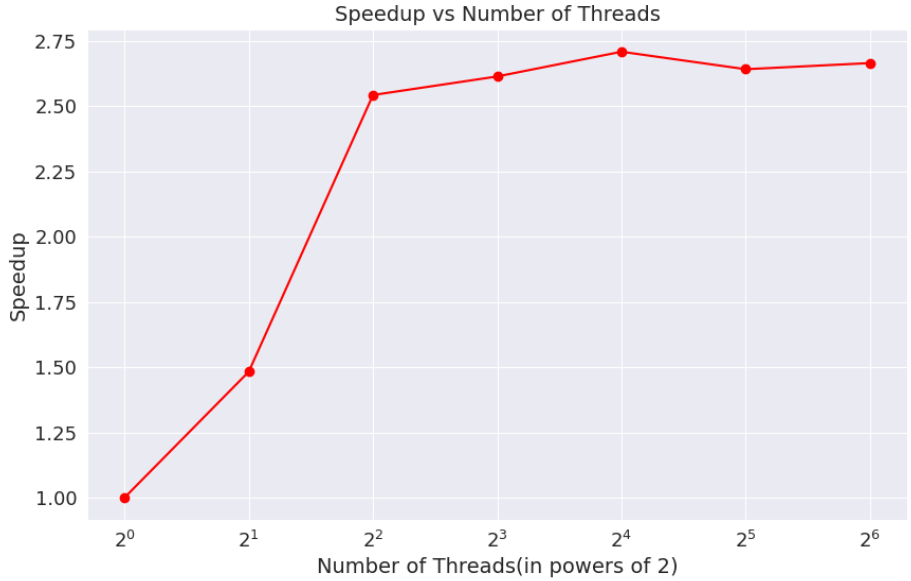


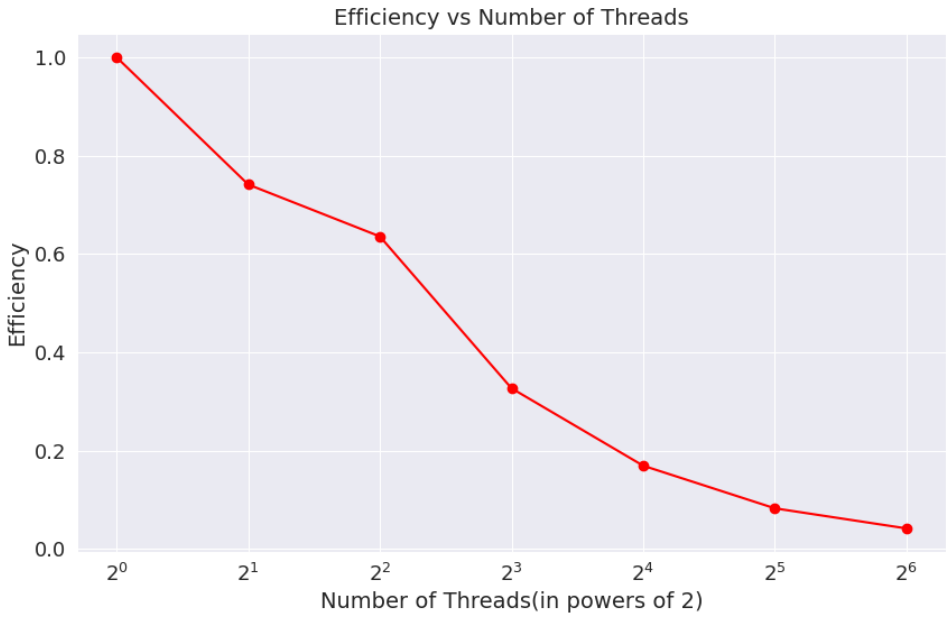
- **Following are the Speed up and efficiency plots for k = 9 and k' = 6**



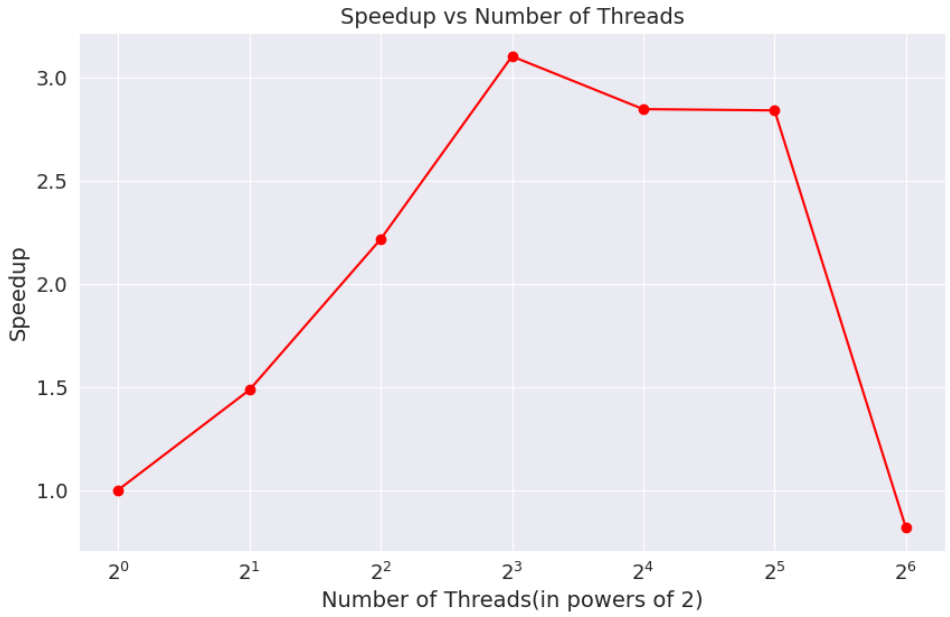


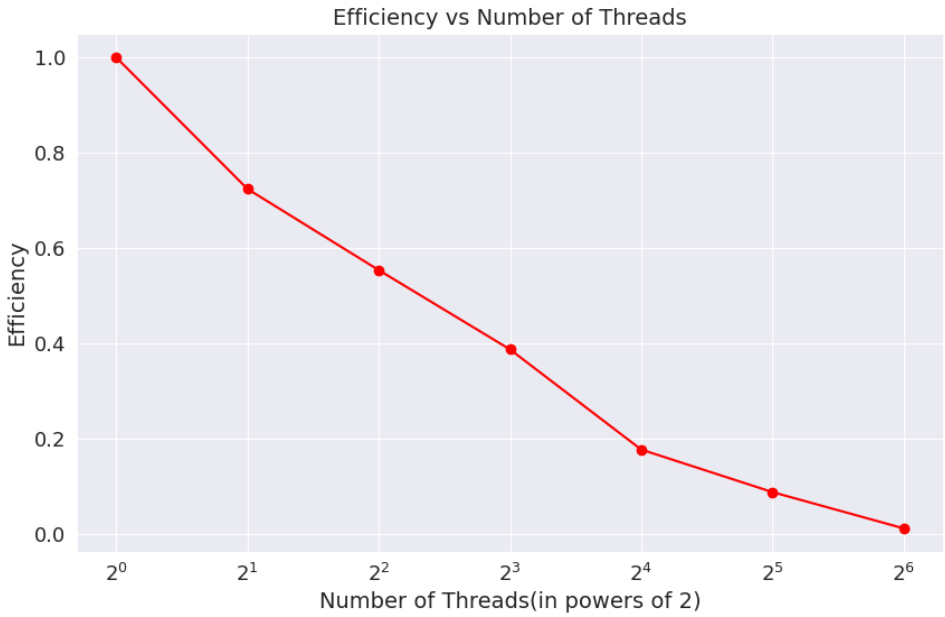
- **Following are the Speed up and efficiency plots for k = 10 and k' = 6**





- **Following are the Speed up and efficiency plots for k = 11 and k' = 6**





Observation for parallel performance by varying k values:

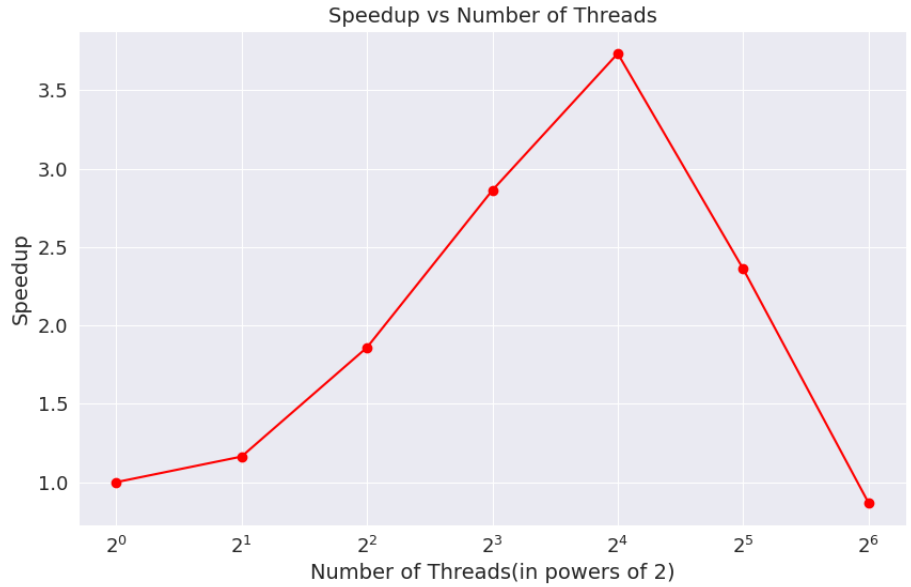
At the outset, we set the lower value of k to 4, which corresponded to a matrix size of 16 x 16, but we did not observe any significant speedup, and the efficiency was quite low. This led us to conclude that parallelized Strassen's multiplication is not particularly effective for smaller matrices. However, as we increased the value of k to 8, 9, 10, and 11, we observed that the execution time increased, likely due to the increased number of recursive calls and overall computation time involved. This is evident from the execution time values presented for the different matrix sizes.

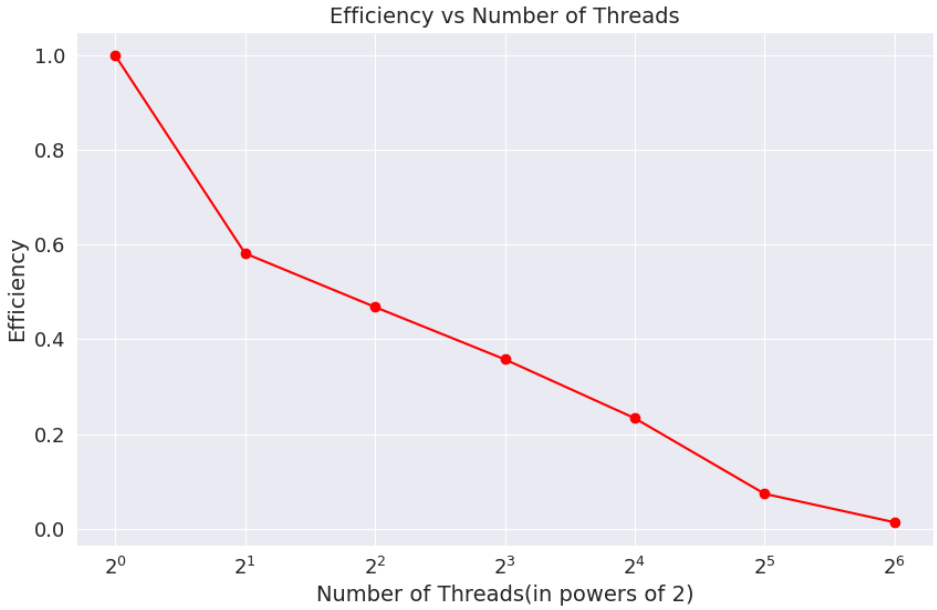
As the matrix size increases, we observe that Strassen's parallel code becomes increasingly effective, resulting in better speedup. Specifically, for values of k equal to 8, 9, 10, and 11, we found that increasing the number of threads by a factor of 2 led to successful parallelization of the code, with a gradual decrease in execution time and increase in speedup. The speedup graphs presented earlier for k' = 6 demonstrate how effective the parallel performance can be, with speedup increasing gradually as k is increased from 8 to 11. However, we also noted that the efficiency plot exhibited a decreasing trend as the number of threads increased.

Now we have fixed the value of k and we are changing the value of terminal matrix k' from k' = 2 ,4, 5 ,6, 7

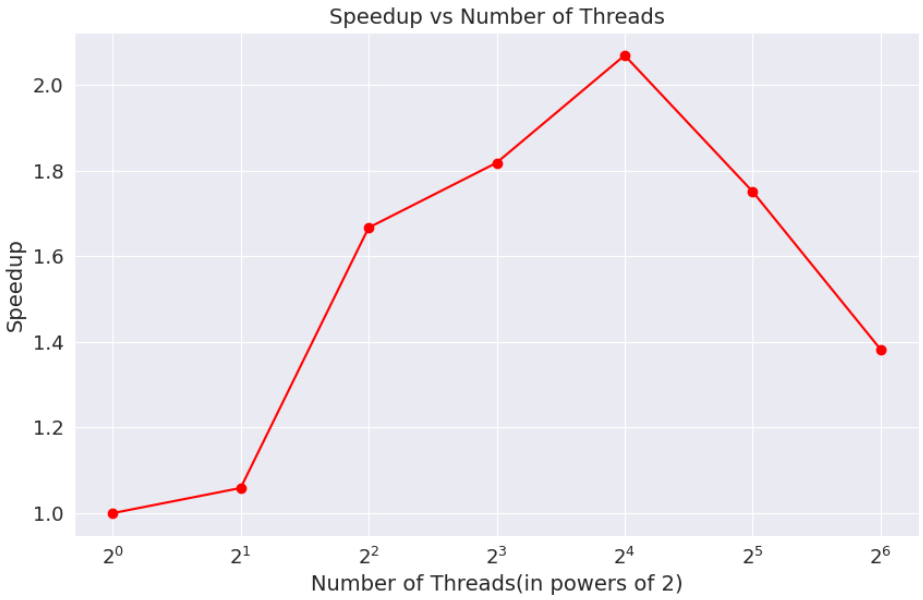
In this section different experiments were conducted to observe the trend of execution time and parallel performance by varying k' values for matrix size k-8 and matrix size k-9.

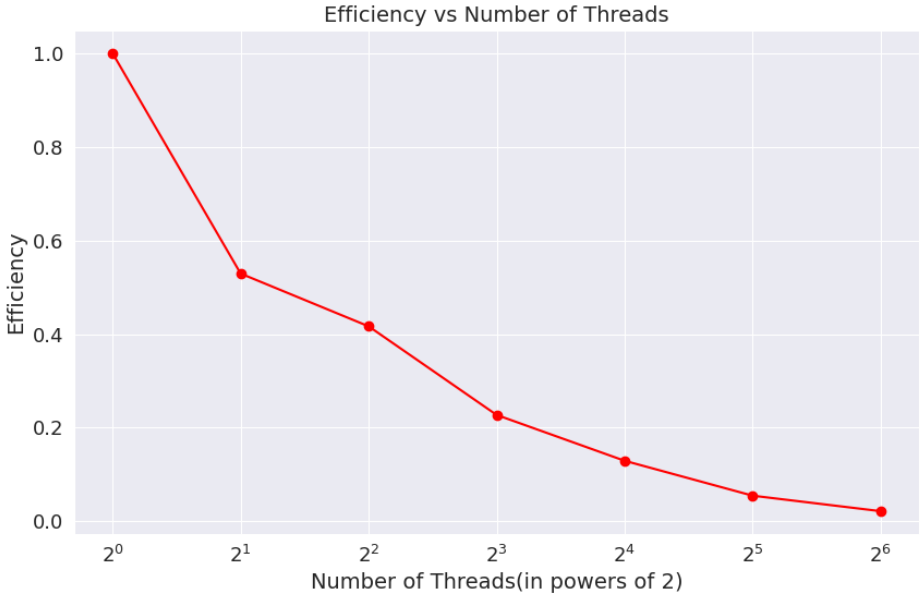
- **Following are the Speed up and efficiency plots for k = 8 and k' = 2**



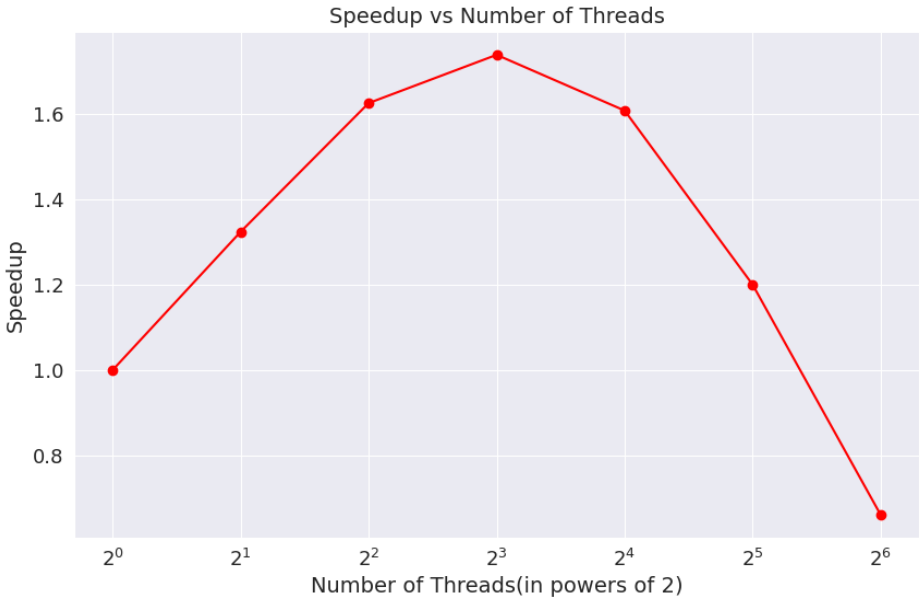


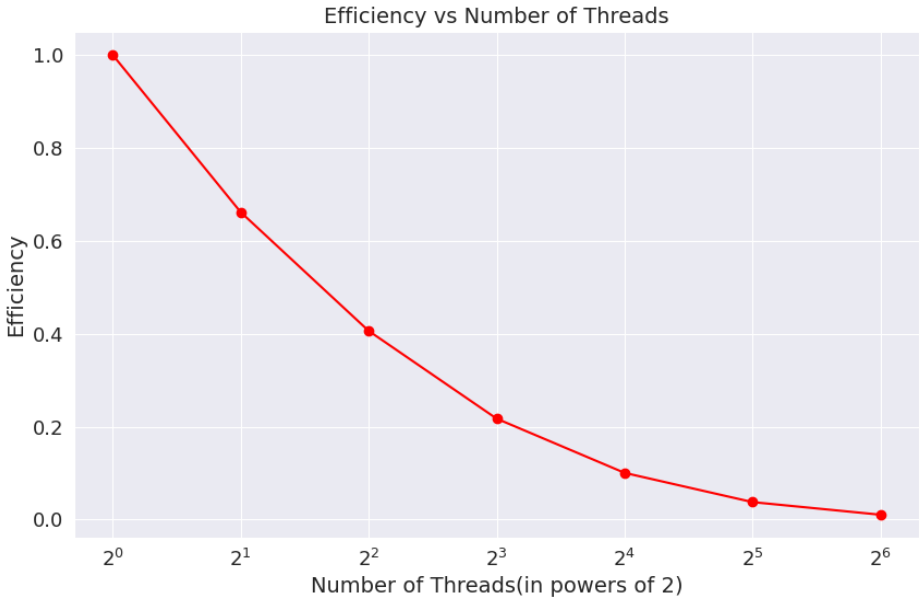
- **Following are the Speed up and efficiency plots for k = 8 and k' = 4**



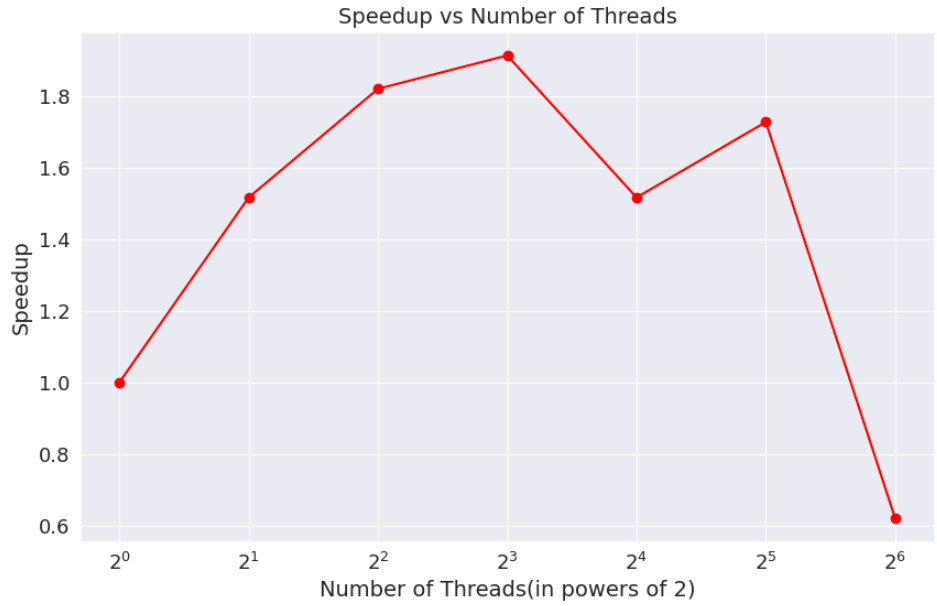


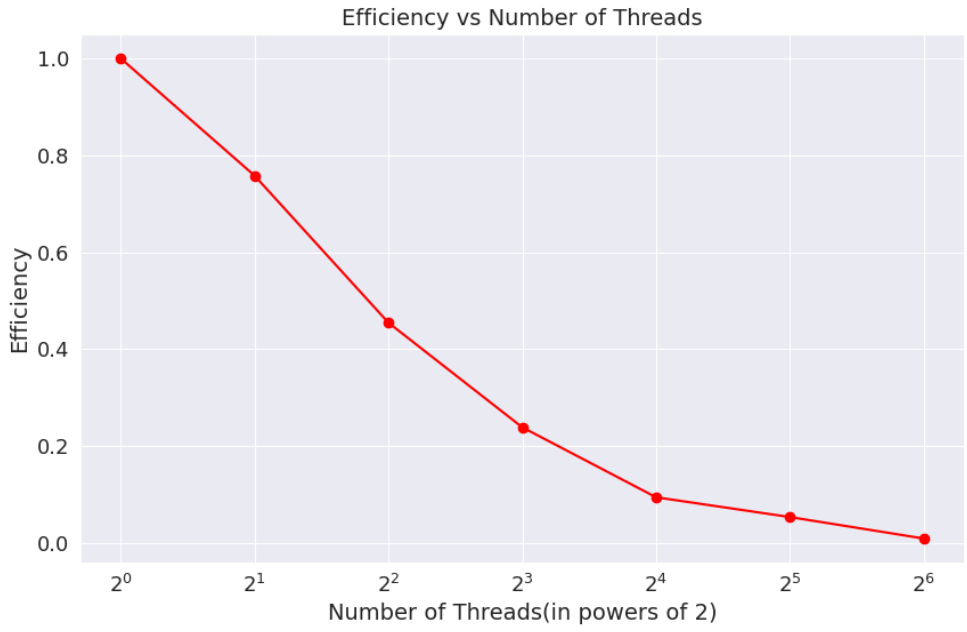
- **Following are the Speed up and efficiency plots for k = 8 and k' = 5**





- **Following are the Speed up and efficiency plots for k = 8 and k' = 6**





Observation and analysis for changing values of terminal matrix size k':

When we set k equal to 8 and increased the value of k', we observed a decrease in execution time and an increase in speedup. This was likely due to the fact that with higher values of k', the number of levels for

normal multiplication increases, and this part of the algorithm can be more efficiently parallelized as compared to the parallelization of recursive calls using pragma tasks. The use of tasks in recursive calls can lead to overhead, and so the parallelization of normal multiplication is more effective than Strassen's approach in this case. This results in a gradual decrease in execution time and an improvement in speedup.

Design choices to improve the parallel performance of the code:

- The project parallelized normal matrix multiplication and recursive Strassen's multiplication for better parallel performance. # pragma omp parallel for was used to parallelize two outer for loops of normal multiplication.

- For levels beyond k' to k where k> k' recursive calls are being made to calculate the value of M1, M2, M3 , M4 ,M5 ,M6 and M7. Here task construct has been used since it defines an explicit task for every recursive call. Since its first private by default for each thread will take its own private variable as it was initially. However, it was found that the task construct did parallelize only the call of the Strassen's multiplication. On successive recursive calls when omp_get_num_threads were used to print only 1 thread was printed showing the task doesn't support nested parallel regions by default. When a nested parallel region was enabled, so that the thread team remained the same in each recursive call a high jump in the execution time was observed due to overhead due to the presence of a large number of concurrent threads. This was decreasing the speed up. So to get best parallel performance omp nested calls can be kept as 0 or ignored as it is 0 by default.

Insights developed after working on this project:

- Using threads for parallelization can improve execution time, but selecting the right number of threads is crucial to avoid overhead and maximize speedup. A range of thread selections was analyzed for optimal parallel performance.
- The hybrid approach of combining normal matrix multiplication with Strassen's recursive approach is advantageous in achieving optimized parallel performance. Tuning the values of k and k' is crucial. Recursive calls creating tasks can result in significant overhead, so tuning the threshold for recursion termination is important.
- Third insight I took from the project is that parallelizing Strassen's multiplication is preferred only for sufficiently larger matrix sizes. The execution time for parallel parts for small matrix size was high and hence no speed up as such.

Brief description on how to compile and execute the code:

To compile and execute below steps are used.

- Initially load the Intel software stack by running *module load intel*.
- To compile the code in non-dedicated mode the command *icc -qopenmp -o strassen.exe strassen.c*
- *./strassen.exe 11 6 5* = runs the code by passing 3 arguments where the first input 11 in this case refers to k (size of matrix is 2^{11}), second argument refers to size of terminal matrix k' (size is $2^{k'}$) and third argument is the number of threads(5 in this case).

- To compile and run in dedicated mode, a batch file needs to be submitted using the command *sbatch strassen.grace_job.*